Dynamic Memory Allocation

CPSC 310 - Programming Languages

Dynamic Memory Allocation

- Programmers use dynamic memory allocators (such as `malloc`) to acquire virtual memory (VM) at run time.
  - For data structures where the size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the heap.

Dynamic Memory Allocation

- Allocator maintains the heap as a collection of variable sized blocks, which are either allocated or free.
- Types of allocators
  - Explicit allocator: application allocates and frees space (for example, `malloc` and `free` in C)
  - Implicit allocator: application allocates, but does not free space (for example, `new` and garbage collection in Java)
- This lecture: explicit memory allocation

The `malloc` Package

- `void *malloc(size_t size)`
  - Success: returns a pointer to a memory block of at least `size` bytes aligned to a 16-byte boundary (on x86-64); if `size == 0`, returns `NULL`
  - Unsuccessful: returns `NULL` and sets `errno` to `ENOMEM`
- `void free(void *p)`
  - Returns the block pointed at by `p` to pool of available memory
  - `p` must come from a previous call to `malloc`, `calloc`, or `realloc`
- Other functions:
  - `calloc`: version of `malloc` that initializes allocated block to zero
  - `realloc`: changes the size of a previously allocated block
  - `sbrk`: used internally by allocators to grow or shrink the heap

malloc Example

```
void foo(long n) {
    long i, *p;
    /* Allocate a block of n longs */
    p = (long *) malloc(n * sizeof(long));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }
    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;
    /* Do something with p */
    ...
    /* Return allocated block to the heap */
    free(p);
}
```

Constraints

- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`'d block
- Explicit Allocators
  - Cannot control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
  - Must allocate blocks from free memory
  - Must align blocks to satisfy alignment requirements
  - Can manipulate and modify only free memory
  - Cannot move the allocated blocks once they are `malloc`'d

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

- Goals: maximize throughput and peak memory utilization
  - these goals are often conflicting
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations per second

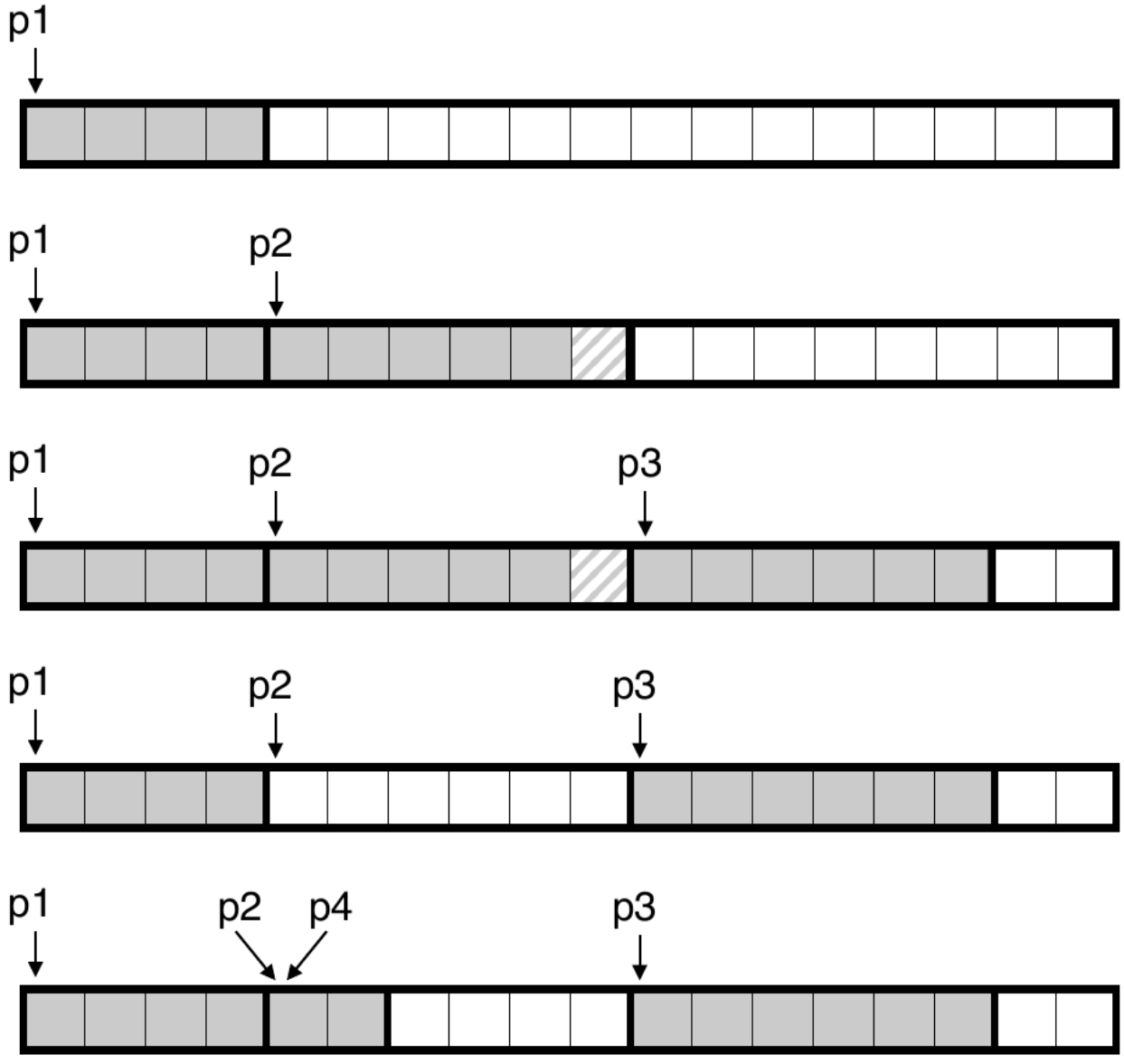
Performance Goal: Minimize Overhead

- Given some sequence of `malloc` and `free` requests:

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

- Definition: aggregate payload P_k
  - `malloc(p)` results in a block with a payload of `p` bytes
  - After request R_k has completed, the aggregate payload P_k is the sum of currently allocated payloads
- Definition: current heap size H_k
  - Assume H_k is monotonically non-decreasing, that is, the heap only grows when the allocator uses `sbrk`
- Definition: Overhead after $k+1$ requests
  - Fraction of heap space not used for program data
  - $O_k = H_k/(max_{i \leq k} P_i) - 1$

`malloc` Heap Visualization Example



Fragmentation

- Fragmentation causes poor memory utilization
- Internal fragmentation: For a given block, internal fragmentation occurs if payload is smaller than block size
  - Caused by
    - overhead of maintaining heap data structures
    - padding for alignment purposes
    - explicit policy decisions (for example, to return a big block to satisfy a small request)
  - Depends only on the pattern of previous requests
- External fragmentation: occurs when there is enough aggregate heap memory, but no single free block is large enough
  - Amount of external fragmentation depends on the pattern of future requests (difficult to measure)

Implementation Issues

- How do we know how much memory to free given only a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is place?
- How do we pick a block to use for allocation – many might fit?
- How do we reuse a block that has been freed?

Knowing How Much to Free

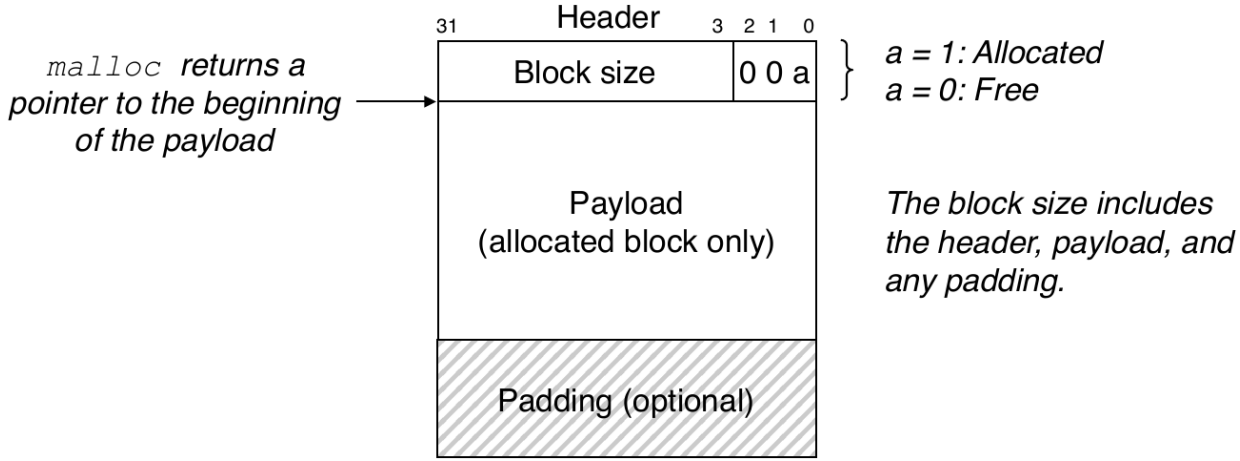
- Standard method
  - Keep the length (in bytes) of a block in the word preceding the block, including the header
  - Requires an extra word for every allocated block

Keeping Track of Free Blocks

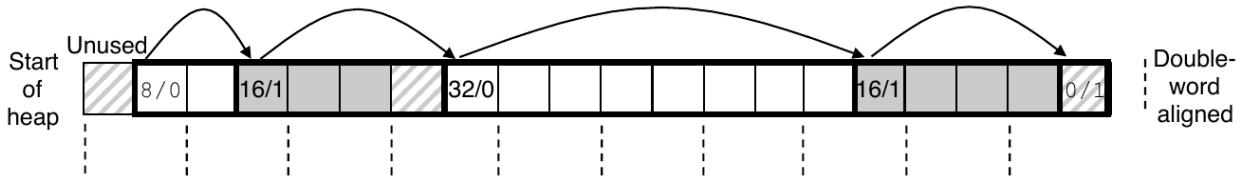
- Method 1: Implicit list using length; links all blocks
  - Need to tag each block as allocated/free
- Method 2: Explicit list among the free blocks using pointers
  - Need space for pointers
- Method 3: Segregated free list
  - Different free lists for different size classes
- Method 4: Blocks sorted by size
  - Can use a balanced tree with pointers within each free block, and the length used as a key

Method 1: Implicit Free List

- For each block we need both size and allocation status
  - Could store this information in two words (wasteful)
- Standard trick
  - When blocks are aligned, some low-order address bits are always zero
  - Instead of storing the always zero bit, use it as an allocated/free flag
  - When reading the size word, the bit must be masked out



Detailed Implicit Free List Example



- Allocated blocks: shaded
- Free blocks: unshaded
- Headers: labeled with "size in words/allocated bit"
  - Headers are at non-aligned positions
  - Payloads are aligned

Implicit List: Data Structures

- Block declaration

```
typedef unint64_t word_t;

typedef struct block {
    word_t header;
    unsigned char payload[0]; // zero length array
} block_t;
```

- Getting payload from block pointer

```
return (void *) (block->payload);
```

- Getting header from payload

```
return (void *) ((unsigned char *) bp
                 - offsetof(block_t, payload));
```

Implicit List: Header access

- Getting allocated bit from header

```
return header & 0x1;
```

- Getting size from header

```
return header & ~0xfL;
```

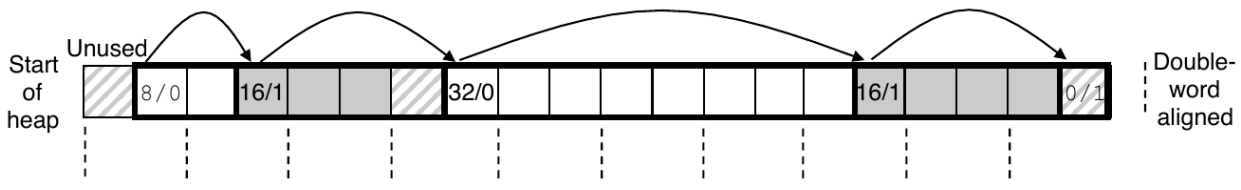
- Initializing header

```
block->header = size | alloc;
```

Implicit List: Traversing the List

- Find next block

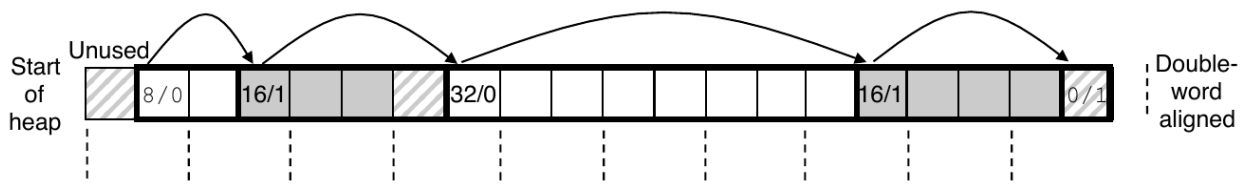
```
static block_t *find_next(block_t *block) {
    return (block_t *) ((unsigned char *) block
                        + get_size(block));
}
```



Implicit List: Finding a Free Block

- Search list from beginning and choose first free block that fits (including space for the header)

```
static block_t *find_fit(size_t asize) {
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block))
    {
      if (!(get_alloc(block)) && (asize <= get_size(block)))
          return block;
    }
    return NULL; // No fit found
}
```

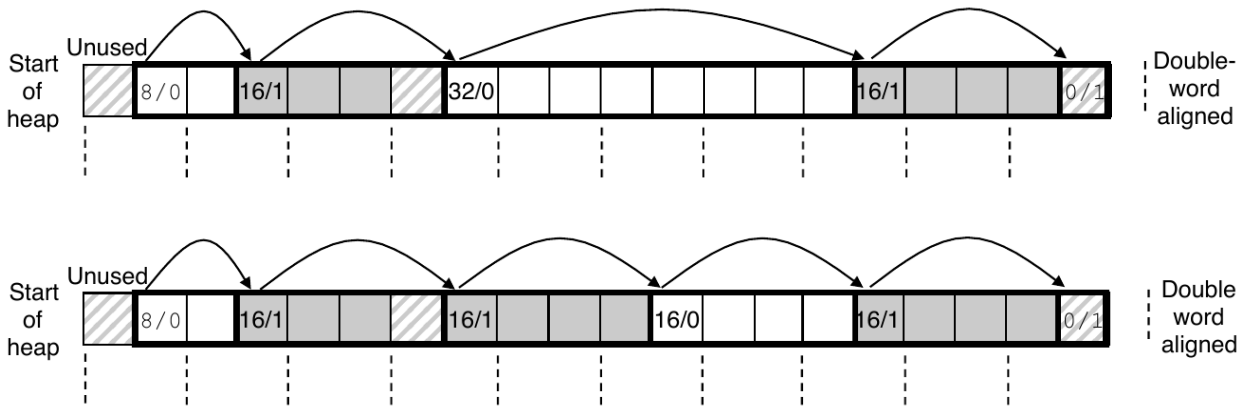


Implicit List: Finding a Free Block

- First fit:
  - Search list from the beginning and choose the first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at the beginning of the list
- Next fit:
  - Like first fit, but search the list starting where the previous search finished
  - Should often be faster than first fit since it avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list and choose the best free block: fits with the fewest bytes left over
  - Keeps fragments small; usually improves memory utilization
  - Will typically run slower than first fit
  - Still a greedy algorithm; no guarantee of optimality

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block



Implicit List: Splitting Free Block

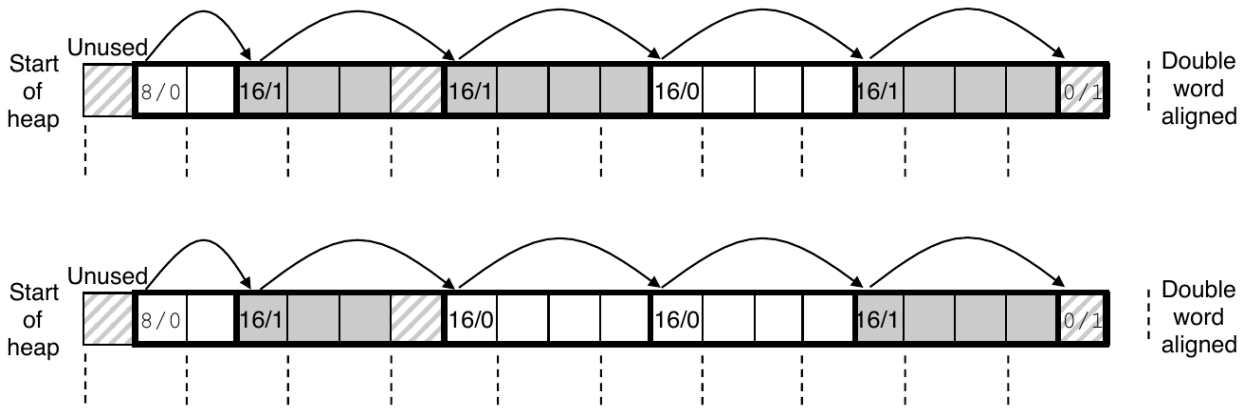
```
// Warning: This code is incomplete

static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next,
                     block_size - asize, false);
    }
}
```

Implicit List: Freeing a Block

- Simplest implementation:
  - Need to clear the "allocated" flag
  - But, can lead to "false fragmentation"

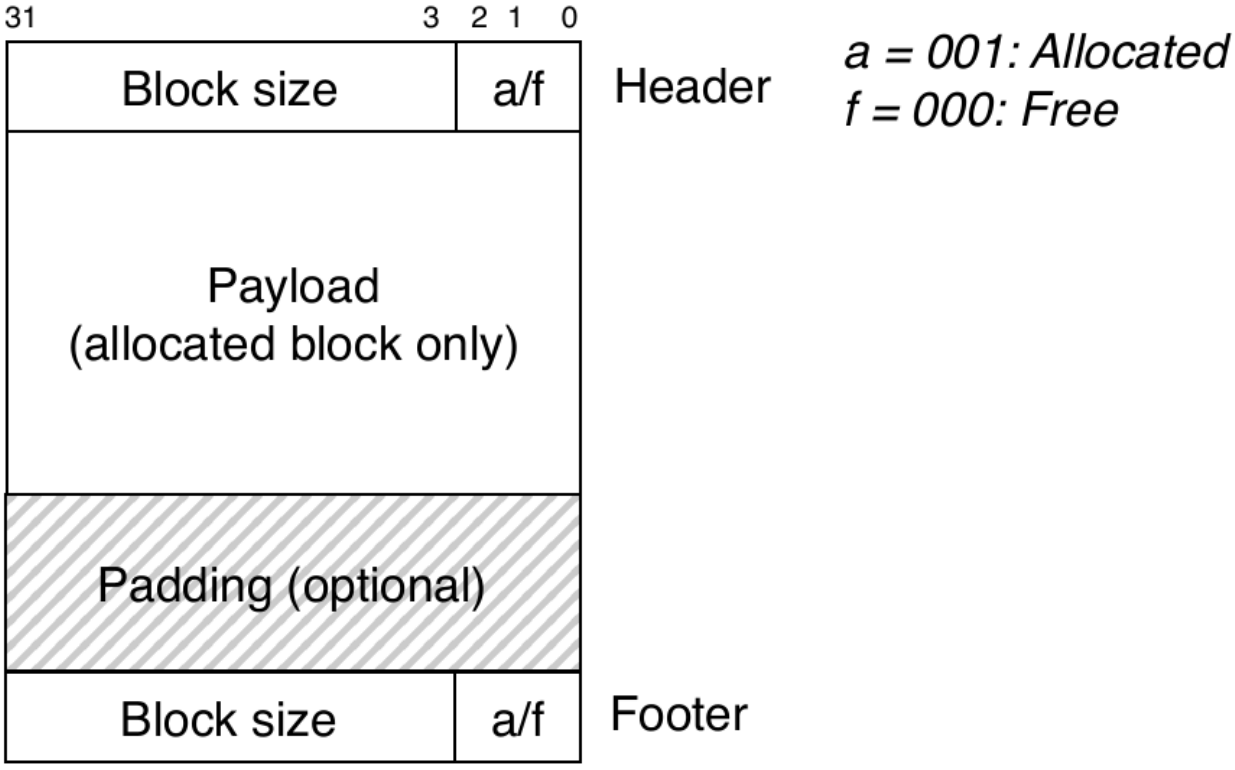


Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
- Coalesce with next block
  - Simple because of forward search
- How do we coalesce with previous block?
  - How do we know where it starts?
  - How can we determine whether it is allocated?

Implicit List: Bidirectional Coalescing

- Boundary tags
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique



Implementation with Footers

- Locating footer of current block

```
const size_t dsize = 2 * sizeof(word_t);

static word_t *header_to_footer(block_t *block) {
    size_t asize = get_size(block);
    return (word_t *) (block->payload
                       + asize - dsize);
}
```

- Locating footer of previous block

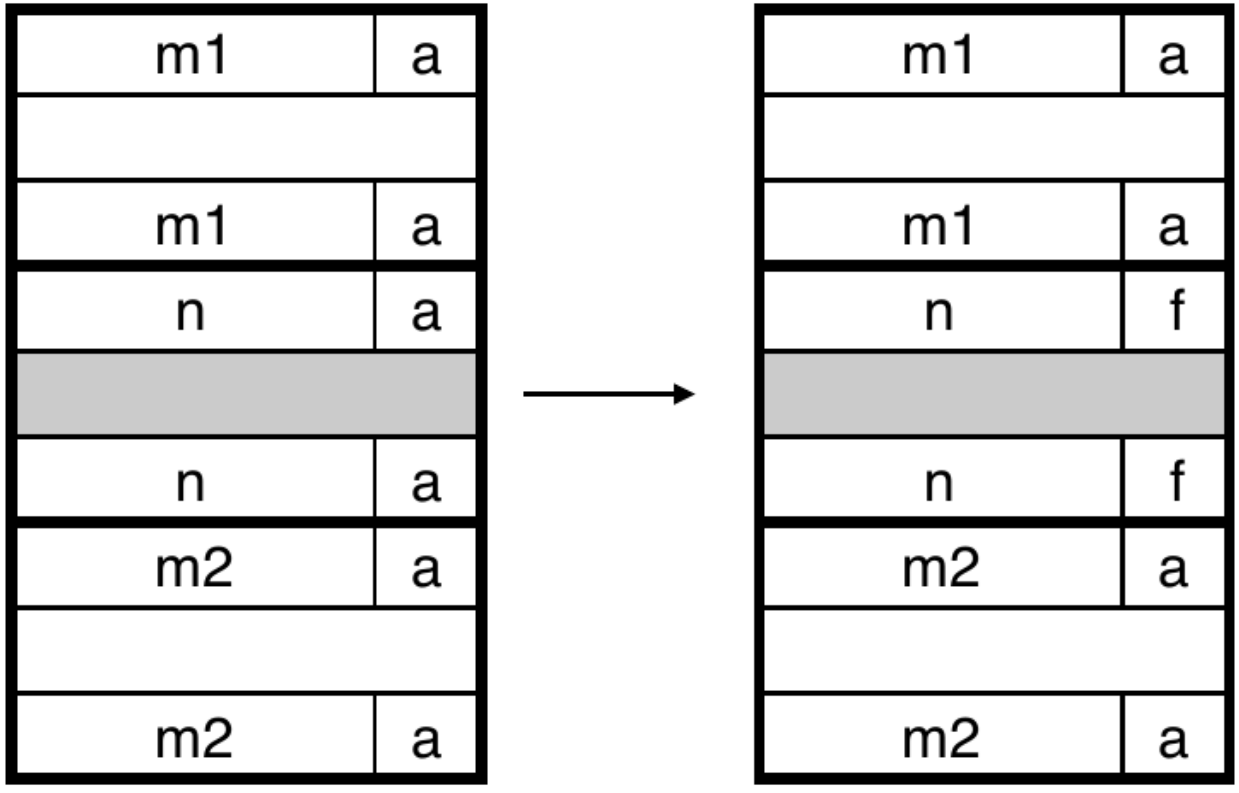
```
static word_t *find_prev_footer(block_t *block) {
    return &(block->header) - 1);
}
```

Splitting Free Block: Full Version

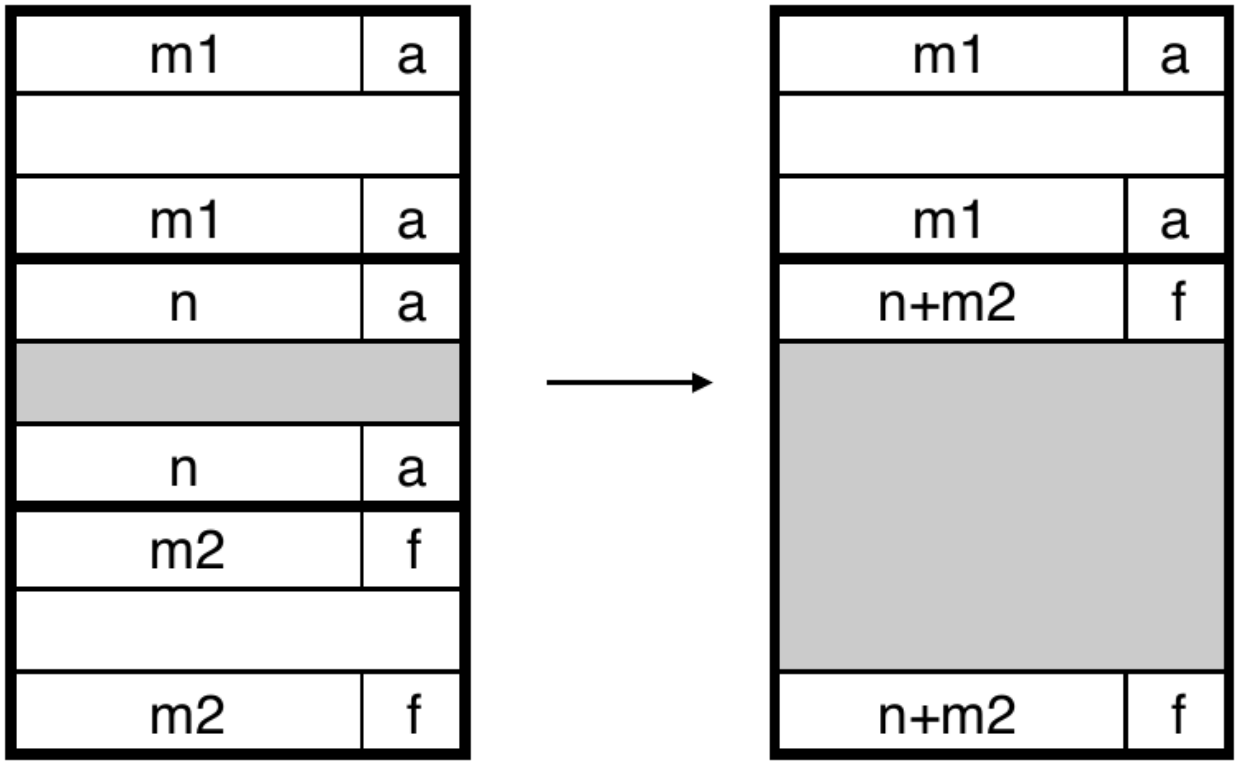
```
static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false)
        write_footer(block_next, block_size - asize, false)
    }
}
```

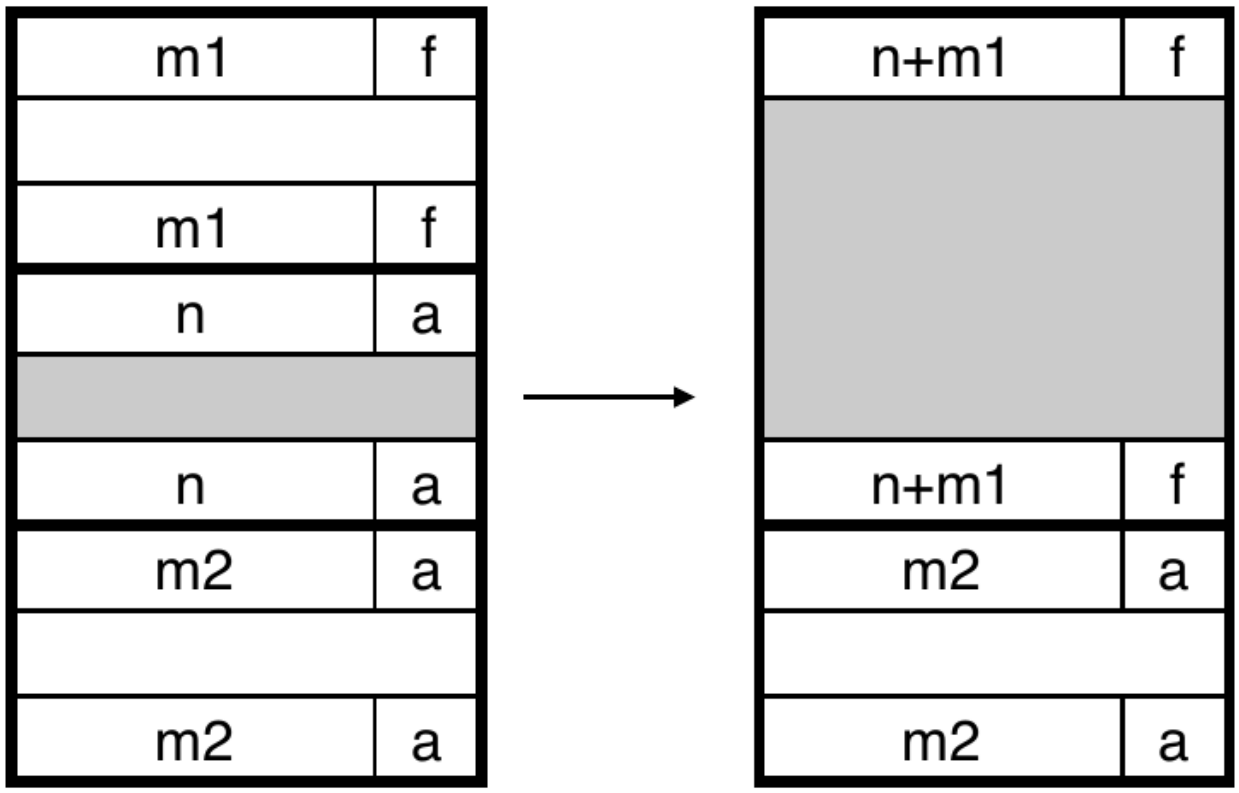
Constant Time Coalescing (Case 1)



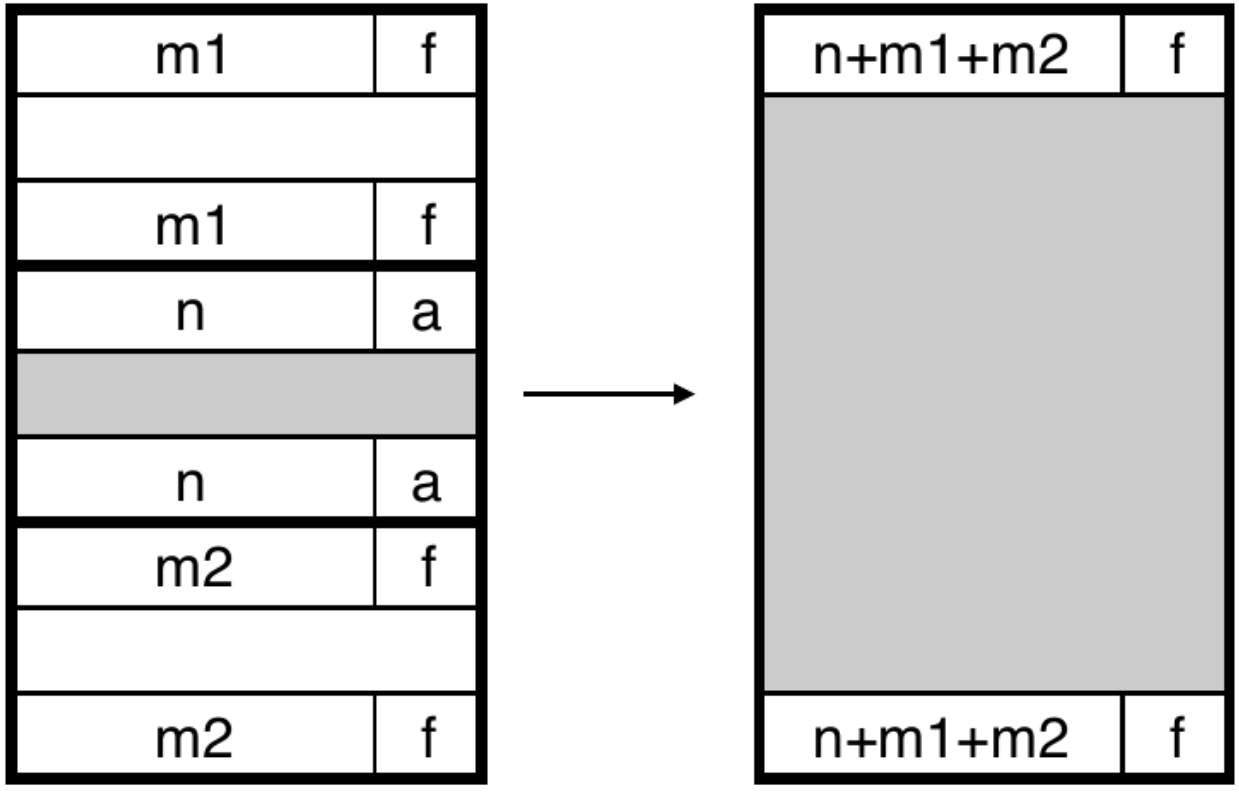
Constant Time Coalescing (Case 2)



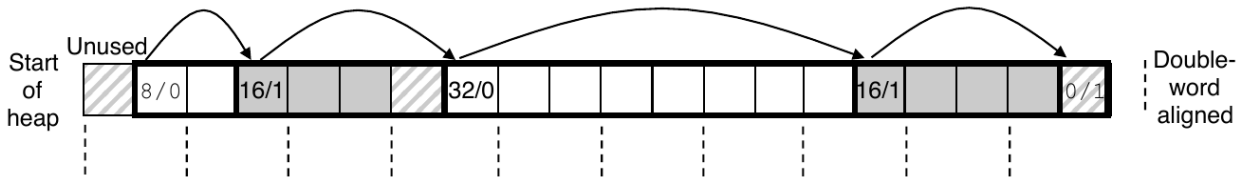
Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Heap Structure



- Dummy footer before first header
  - Marked as allocated
  - Prevents accidental coalescing when freeing first block
- Dummy header after last footer
  - Prevents accidental coalescing when freeing final block

Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);
```

Top-Level Free Code

```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

No Boundary Tag for Allocated Blocks

- Boundary tag needed only for free blocks
- When sizes are multiples of 16, have 4 spare bits
- Header: Use 2 bits (address bits always zero due to alignment):

  `(prev_block) << 1 | (curr_block)`

Explicit Free Lists

- Maintain list(s) of *free* blocks, not *all* blocks
  - We track only free blocks, so we can use payload area
  - The "next" free block could be anywhere
    - We need to store forward/backward pointers, not just sizes
  - Still need boundary tags for coalescing
    - To find adjacent blocks according to memory order

Freeing With Explicit Free Lists

- Insertion policy: where in the free list do you put a newly freed block?
- Unordered
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end of the free list
  - Pro: simple and constant time
  - Con: studies suggest fragmentation is worse than address ordered
- Address-ordered policy
  - Insert freed blocks so that free list blocks are always in address order:

    $addr(prev) < addr(curr) < addr(next)$

  - Con: requires search

Freeing with a LIFO Policy

- Case 1: allocated $\leftrightarrow$ target $\leftrightarrow$ allocated
  - Insert the freed block at the root of the list
- Case 2: allocated $\leftrightarrow$ target $\leftrightarrow$ free
  - Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list
- Case 3: free $\leftrightarrow$ target $\leftrightarrow$ allocated
  - Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list
- Case 4: free $\leftrightarrow$ target $\leftrightarrow$ free
  - Splice out adjacent successor block, coalesce all three memory blocks, and insert the new block at the root of the list

An Implementation Trick

- Use circular, doubly linked list
- Support multiple approaches with single data structure
- First-fit versus next-fit
  - Either keep free pointer fixed or move as search list
- LIFO versus FIFO
  - Insert as next block (LIFO) or previous block (FIFO)

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of *free* blocks instead of *all* blocks (much faster)
  - Slightly more complicated allocate and free because we need to splice blocks in and out of the list
  - Some extra space for the links (two extra words needed for each block)
    - Does this increase internal fragmentation?

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list
  - Example size classes: 16, 32-48, 64-inf
- Often have separate classes for each small size
- For larger sized: one class for each size $[2^i + 1, 2^{i+1}]$

Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size n:
  - Search appropriate free list for block of size $m > n$ (that is, first fit)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
    - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of n byte from this new memory
  - Place remainder as single free block in appropriate size class

Seglist Allocator (Continued)

- To free a block:
  - Coalesce and place on appropriate list
- Advantages of seglist allocators versus non-seglist allocators (both with first fit)
  - Higher throughput
    - log time for power-of-two size classes versus linear time
  - Better memory utilization
    - First fit search of segregated free list approximates a best fit search of the entire heap
    - Extreme case: giving each block its own size class is equivalent to best fit

Memory Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf("%d", val);
```

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int)); // <-- here
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

- Can avoid by using `calloc`

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

- Can you spot the bug?

Overwriting Memory

- Off-by-one errors

```
char **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) { // <-- here
   p[i] = malloc(M*sizeof(int));
}

char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

Overwriting Memory

- Not checking the max string size

```
char s[8]; // <-- too small
int i;

gets(s);  /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int); // <-- here

   return p;
}
```

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;  // <-- here
   Heapify(binheap, *size, 0);
   return(packet);
}
```

- What gets decremented?
  - Hint: precedence and associativity

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

```
x = malloc(N*sizeof(int));
      <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

Referencing Freed Blocks

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
     ...
    free(head);
    return;
}
```

Dealing With Memory Bugs

- Debugger: `gdb`
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Data structure consistency checker
  - Runs silently, prints message only on error
  - Use as a probe to zero in on error
- Binary translator: `valgrind`
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- `glibc malloc` contains checking code

Implicit Memory Management: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage; application never has to explicitly free memory
- Common in many dynamic languages
- Variants ("conservative" garbage collectors) exist for C and C++

Garbage Collection

- How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if there are no pointers to them
- Must make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers (for example, coercing them to an `int` and then back again)

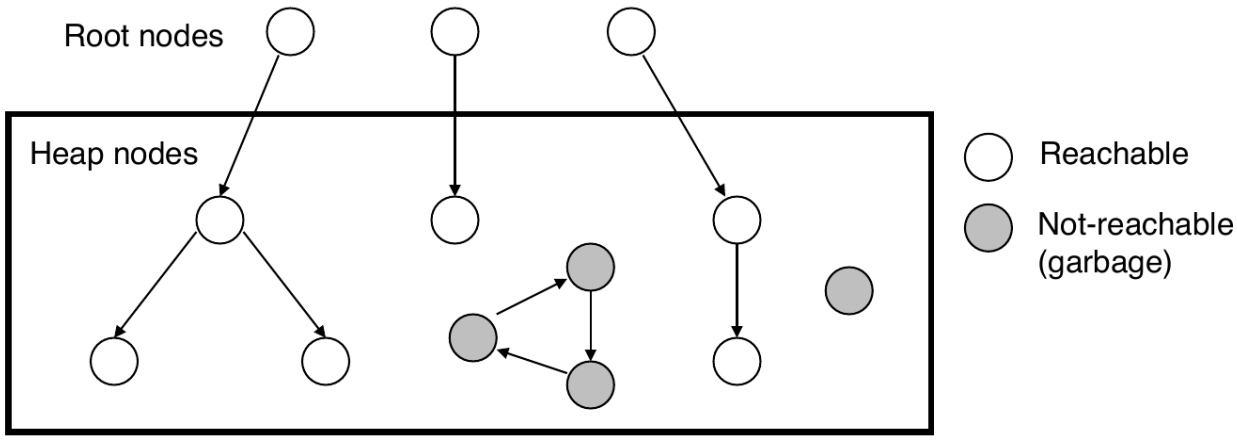
Classical Garbage Collection Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So, focus reclamation work on zones of memory recently allocated

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called **root** nodes (for example, registers, locations on the stack, global variables)

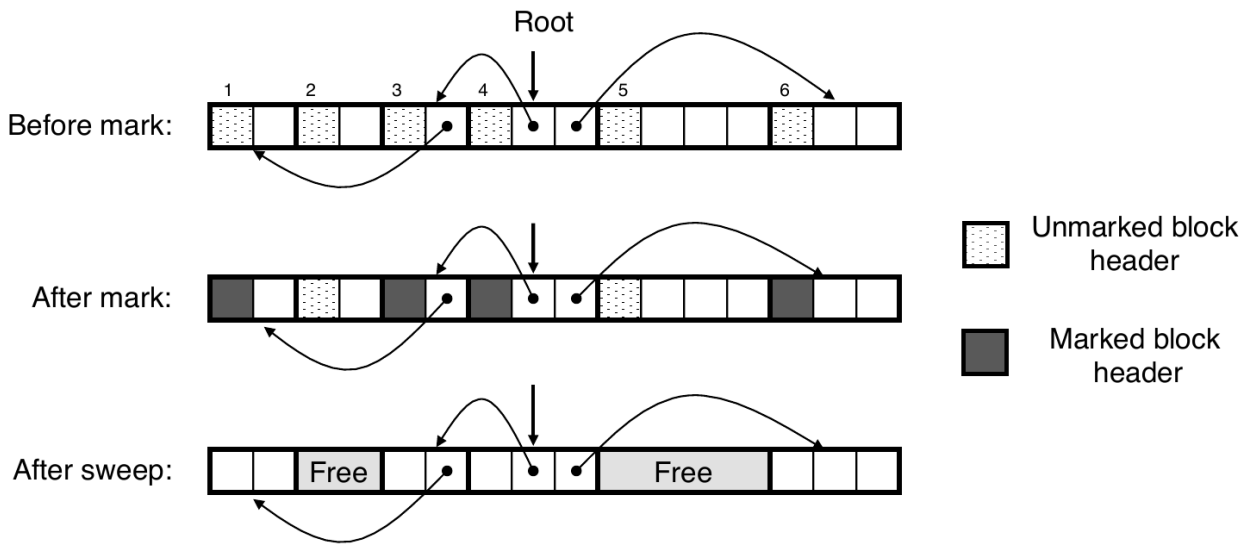
Memory as a Graph



- A node (block) is *reachable* if there is a path from any root to that node
- Non-reachable nodes are *garbage* (cannot be needed by the application)

Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - Use extra *mark bit* in the head of each block
  - Mark: start at roots and set mark bit on each reachable block
  - Sweep: scan all blocks and free blocks that are not marked



Assumptions for a Simple Implementation

- Application
  - `new(n)`: returns pointer to new block with all locations cleared
  - `read(b, i)`: read location `i` of block `b` into register
  - `write(b, i, v)`: write `v` into location `i` of block `b`
- Each block will have a header word
  - addressed as `b[-1]`, for block `b`
  - used for different purposes in different collectors
- Instructions used by the Garbage Collector
  - `is_ptr(p)`: determines whether `p` is a pointer
  - `length(b)`: returns the length of block `b`, not including the header
  - `get_roots()`: returns all the roots

Mark and Sweep Pseudocode

- Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;
    if (markBitSet(p)) return;
    setMarkBit(p);
    for (i=0; i < length(p); i++)
        mark(p[i]);
    return;
}
```

Mark and Sweep Pseudocode

- Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```